HoGent

BEDRIJF
EN
ORGANISATIE

Singleton

Unieke objecten waarvoor
slechts 1 instantie bestaat

Singleton

- Er bestaan veel objecten waar we maar 1 instantie van nodig hebben
  - Thread pools, caches, objecten voor het loggen, objecten die fungeren als device drivers voor apparaten als printers en grafische kaarten,…
  - Meer dan 1 instantie aanmaken van deze objecten garandeert problemen
- Singleton garandeert dat er maar één en niet meer dan één instantie van een bepaalde klasse bestaat
- Het levert 1 globaal toegangspunt
- Je maakt het aan wanneer nodig

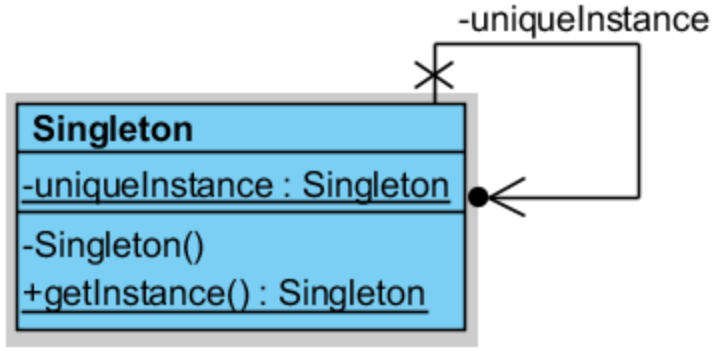
De implementatie ontrafelen

- Garandeert dat een klasse maar één instantie heeft en biedt één globaal toegangspunt daartoe

```
public class Singleton {

  private static final Singleton uniqueInstance
        = new Singleton();

  //hier komen nog andere nuttige instantievariabelen

  private Singleton() {
  }

  public static Singleton getInstance() {
    return uniqueInstance;
  }

  //andere nuttige methoden
}
```

Dit is een static variabele om de enige instantie van de klasse Singleton te bewaren.

De constructor wordt private gedeclareerd en bevat geen parameters. Enkel Singleton kan deze klasse instantiëren



Lazy loading

- Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used. The opposite of lazy loading is Eager Loading.

Lazy loading - Wikipedia, the free encyclopedia. (n.d.). Retrieved February 04, 2015, from http://en.wikipedia.org/wiki/Lazy_loading

```
public class SnookerPlayer {

  private SnookerCue snookerCue;

  public SnookerCue getSnookerCue() {
    if (snookerCue == null) {
      snookerCue = new SnookerCue();
    }
    return snookerCue;
  }
  //...
}
```

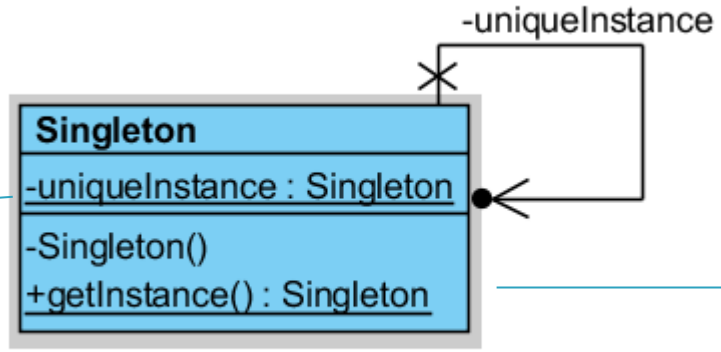
Singleton & Lazy instantiation

De klassieke implementatie

```
public class Singleton {

    private static Singleton uniqueInstance;

      //hier komen andere nuttige instantievariabelen
    private Singleton() {
    }

    public static Singleton getInstance() {
      if (uniqueInstance == null) {
        uniqueInstance = new Singleton();
      }
      return uniqueInstance;
    }

      //andere nuttige methoden
}
```

Singleton – creëren van objecten

Het Singleton Pattern garandeert dat een klasse slechts één instantie heeft, en biedt een globaal toegangspunt ernaartoe.



Bevat de enigste instantie van het Singleton

De methode getInstance is static, dus te benaderen via Singleton.getInstance(). De toegang is even eenvoudig als naar een globale variabele, maar we hebben de voordelen van de late instantiatie van het Singleton

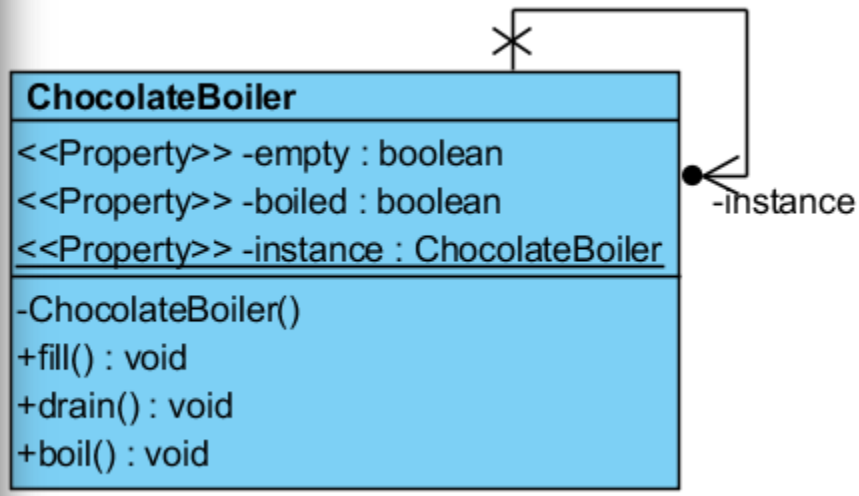
Voorbeeld : de chocoladefabriek

- Alle moderne chocoladefabrieken gebruiken computergestuurde chocoladeboilers. De boiler verwarmt een mengsel van chocolade en melk, en draagt dit vervolgens over aan de volgende fase van het productieproces voor chocoladerepen.

- Op de volgende slide staat de controllerklasse voor de industriële chocoladeboiler. Bekijk de code. Je zult zien dat ze erg hun best hebben gedaan om ervoor te zorgen dat er geen gekke dingen gebeuren, zoals het afvoeren van 2500 liter onverwarmd mengsel, het vullen van een boiler terwijl hij al vol zit of het verwarmen van een lege boiler!

Voorbeeld : de chocoladefabriek

```
public class ChocolateBoiler {

  private boolean empty;
  private boolean boiled;

  private static ChocolateBoiler instance;

  private ChocolateBoiler() {
    empty = true;
    boiled = false;
  }

  public static ChocolateBoiler getInstance() {
    if (instance == null) {
      instance = new ChocolateBoiler();
    }
    return instance;
  }
```



efabriek

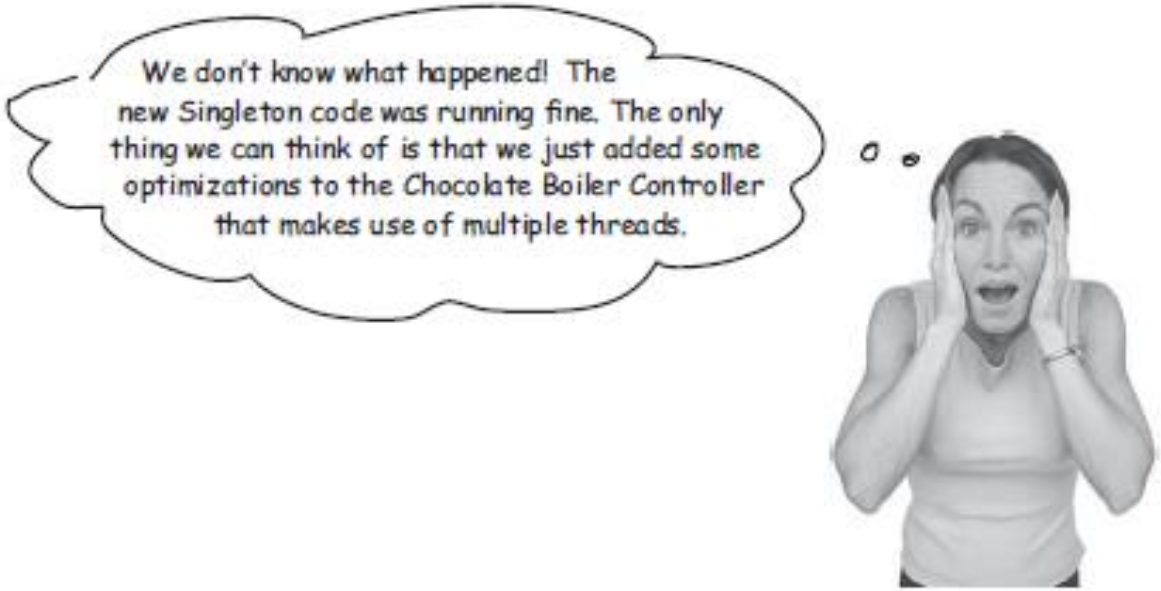
```
public void fill() {
   if (isEmpty()) {
      empty = false;
      boiled = false;
      // vul de boiler met een mengsel van melk en chocolade
   }
}
public void drain() {
   if (!isEmpty() && isBoiled()) {
      // voer de verwarmde melk en chocolade af
      empty = true;
   }
}
public void boil() {
   if (!isEmpty() && !isBoiled()) {    // verwarm de inhoud
      boiled = true;
   }
}
```

```
public boolean isEmpty() {
   return empty;
}

public boolean isBoiled() {
   return boiled;
}
```

Multithreading

- Houston, we have a problem!

2500 liters of spilled milk and chocolate



Multithreading

- Stel we hebben twee threads die beide deze code uitvoeren. Het is jouw taak om de JVM te spelen en te bepalen of het mogelijk is dat de twee threads met verschillende boilerobjecten te maken kunnen hebben.

```
private static ChocolateBoiler instance;

public static ChocolateBoiler getInstance() {
   if (instance == null) {
      instance = new ChocolateBoiler();
   }
   return instance;
}
```

Multithreading

```
private static ChocolateBoiler instance;

public static ChocolateBoiler getInstance() {
   if (instance == null) {
      instance = new ChocolateBoiler();
   }
   return instance;
}
```

- Gebruik codemagneten om te bestuderen hoe de code door elkaar kan lopen om 2 boilerobjecten te maken

Thread 1	Thread 2	Waarde van instance

Multithreading

- Oplossing

```
private static ChocolateBoiler instance;

public static ChocolateBoiler getInstance() {
   if (instance == null) {
      instance = new ChocolateBoiler();
   }
   return instance;
}
```

Thread 1	Thread 2	Waarde van instance
`public static ChocolateBoiler getInstance() {`		null
	`public static ChocolateBoiler getInstance() {`	null
`if (instance == null) {`		null
	`if (instance == null) {`	null
`instance = new ChocolateBoiler();`		<object1>
`return instance;`		<object1>
	`instance = new ChocolateBoiler();`	<object2>
	`return instance;`	<object2>

Multithreading oplossing (zonder lazy loading)

- Maak meteen een instantie ipv een uitgestelde instantie.

```
public class Singleton {

   private static final Singleton uniqueInstance
         = new Singleton();

   private Singleton() {
   }

   public static Singleton getInstance() {
      return uniqueInstance;
   }

}
```

Met deze aanpak vertrouwen we op de JVM om een unieke instantie van de Singleton te maken wanneer de klasse wordt geladen. De JVM garandeert dat de instantie gemaakt wordt voordat een thread toegang krijgt tot de static variabele uniqueInstance.

Multithreading oplossing (met lazy loading)

- Maak pas een instantie wanneer er voor het eerst naar gevraagd wordt.

```
public class Singleton {

  private static Singleton uniqueInstance;

  private Singleton() {
  }

  public static synchronized Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
}
```

Nadeel: synchronisatie is duur! (Kan performantie met factor 100 reduceren) Nadat de code voor de eerste maal is doorlopen, is synchronisatie volledig overbodig!

Singleton voorbeelden

- Calendar calendar = Calendar.getInstance();
  - “Gets a calendar using the default time zone and locale.”

- System.out
  - Het PrintStream object verbonden met de enige standaard outputstream van de applicatie

- System.in
  - Het InputStream object verbonden met de enige standaard inputstream van de applicatie